Overcoming Resolution-Based Lower Bounds for SAT Solvers

DoRon B. Motter and Igor L. Markov
University of Michigan, EECS
1301 Beal Ave
Ann Arbor, MI 48109-2122
{dmotter,imarkov}@eecs.umich.edu

ABSTRACT

Many leading-edge SAT solvers are based on the Davis-Putnam procedure or the Davis-Logemann-Loveland procedure, and thus on unsatisfiable instances they can be viewed as attempting to find refutations by resolution. Therefore, exponential lower bounds on the length of resolution proofs also apply to such solvers. Empirical performance of DLL-based solvers on SAT instances from the pigeonhole and Urquhart family are consistent with this expectation.

Our work explores an entirely different approach to SAT solving that does not have this drawback. A bare-bones implementation of our algorithm, reported earlier, was able to refute pigeonhole instances in polynomial time without explicitly using symmetries, and this empirical result is backed up by an analytical proof. In this work, we show how to extend Compressed-BFS to perform Boolean Constraint Propagation, part of all practical, complete SAT solvers. Unlike DLL-based solvers, our empirical results show that full BCP offers marginal improvements in runtime.

1. INTRODUCTION

State-of-the-art, complete SAT solvers are usually based on the Davis-Logemann-Loveland (DLL) search procedure [8]. DLL is a backtracking algorithm with several extensions, but its runtime on unsatisfiable instances is bounded from below by the length of resolution proofs. This fact can be combined with known exponential lower bounds for resolution proofs of certain SAT instance families, such as the pigeonhole instances (`hole-n`). [9, 5, 17]. The result is that any implementation of the DLL algorithm must require exponential time to refute `hole-n` instances [5]. Indeed, the leading-edge SAT solvers Chaff [12] and GRASP [15] empirically take exponential time on these instances. In addition the *OBDD-apply* approach to SAT must also take exponential time on pigeonhole instances [5].

Recent practical work in SAT has primarily focused on implementation details used for the DLL procedure; a different avenue of research is to look for new SAT algorithms whose complexity is not lower-bounded by resolution. Put differently, we are searching for SAT solvers which lead to different classes of tractable SAT instances. To this end, we point out that the recently reported Compressed-BFS algorithm [13] empirically solves pigeonhole instances (`hole-n`) in polynomial time. This observation is supported by an analytical proof [14]. While we do not claim resolution is subsumed by Compressed-BFS, there is an infinite family of instances where Compressed-BFS exponentially outperforms resolution, and thus lower-bounds based on resolution do not apply.

While the original implementation, Cassatt [13], efficiently solves only some classes of benchmarks, we believe that this is not a fundamental limitation. Highly refined DLL implementations contain many performance enhancing features which do not directly translate to the type of search used in Compressed-BFS. In this work, we observe that Boolean Constraint Propagation (BCP) is an integral feature of many complete SAT solvers, and develop it in the context of Compressed-BFS.

The remaining part of this paper is organized as follows. Section 2 reviews the necessary background, while the Compressed-BFS algorithm is described in Section 3. In Section 4 we introduce BCP into Compressed-BFS. Empirical results are presented in Section 5, while conclusions and our ongoing research are described in Section 6.

2. BACKGROUND

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of Boolean variables. A *truth assignment* for V is a mapping $t : V \to \{0,1\}$. For any variable $v \in V$, let v and $\bar{v}$ be called *literals*. A *clause* is a set of literals. A clause is *satisfied* by a truth assignment t if at least one of its literals is true under t. A clause is said to be *violated* by a truth assignment t if all of its literals are false under t. A Boolean formula in conjunctive normal form (CNF) can be represented by a set C of clauses.

The implicit representation used in the Compressed-BFS algorithm is dependent on the correspondence between valid partial truth assignments and sets of clauses. Given a set of Boolean variables V, a *partial truth assignment* for V assigns values only to some subset of variables $V' \subseteq V$. A partial truth assignment is *invalid* if it violates some clause.

Given a valid partial truth assignment t, we can classify clauses in a CNF with respect to t as follows.

- *Open* clauses have at least one literal assigned, and no literals true.
- *Satisfied* clauses have at least one literal assigned true.
- *Activated* clauses have at least one literal assigned.
- *Unit* clauses have all but one literal assigned, and are open.

To illustrate the correspondence between sets of *open* clauses and valid partial truth assignments, it is helpful to consider an example.

$$\underbrace{(a+c+d)}_{1}\underbrace{(\bar{b}+e+f)}_{2}\underbrace{(d+\bar{e})}_{3}\underbrace{(\bar{g}+\bar{h})}_{4}$$

We will view this example from the context of a Breadth-First Search: all valid partial truth assignments we will consider will be to the same set of variables. Assume that in this valid partial truth assignment, variables $\{a,b,c,d\}$ have been assigned values. Immediately we know that clause 4 is *not activated*, and also that clause 1 is *satisfied*. This is true since we are considering a valid assignment, and all literals of clause 1 have been assigned values. Then this clause must be satisfied by our assignment. The only clauses in which the actual values assigned matter are those *in the cut*. clauses, 2 and 3. These *cut* clauses straddle a conceptual vertical line separating assigned variables from unassigned ones as shown in Figure 1.

Now consider the specific valid partial assignment t given by $\{a = 1, b = 0, c = 1, d = 0\}$. Given a specific assignment, we can determine which clauses in the *cut* are *open* and which clauses are *satisfied*. Clause 2 is satisfied by t, while clause 3 remains *open*. Since clause 3 has only one free literal, it is in fact *unit*. Note that only clauses in the *cut* have the potential to be open or unit clauses.

Our algorithm relies on the fact that there is a correspondence between a *valid partial truth assignment* and a set of *open* clauses. Storing

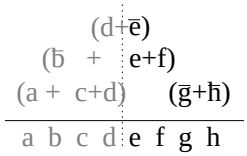

Figure 1: Classification of Clauses

subsets of *open* clauses instead of explicit partial truth assignments is enough information to perform a BFS and determine satisfiability of a formula. Within the context of the Compressed-BFS algorithm, this collection of subsets of *open* clauses is called the *front*.

2.1 Zero-Suppressed Binary Decision Diagrams

A Binary Decision Diagram is defined to be a directed acyclic graph (DAG) with two sink nodes. Each non-sink node has in this graph has a unique label, an integer index, and two outgoing edges. One outgoing edge gets the label 1, while the other outgoing edge gets the label 0. Each outgoing edge can connect only to child nodes at lower levels. Because of this we can represent each node X as a 3-tuple $X\langle n, X_T, X_E\rangle$ where n is the index of the node X, X_T is the node reached after traversing the 1-edge, and X_E is the node reached after traversing the 0-edge. Throughout this work, diagrams will use a solid line to indicate a node's 1-edge, and a dashed line to indicate a node's 0-edge. Each path in the DAG ends in one of two sink nodes, the **0** node and the **1** node. In addition, there is a single root node. The semantics of BDD can be defined recursively by defining the semantics of a given node.

A BDD can be used to encode a collection of sets by encoding this collection's characteristic function. We can evaluate a function represented by a BDD by traversing the DAG beginning at the root node. At each node X, if the variable corresponding to the index of X is true, we traverse along the 1-edge. Otherwise we traverse along the 0-edge. Eventually we will reach either **0** or **1**, indicating the value of the function on this input. For Zero-Suppressed Binary Decision Diagrams, we augment this with the *Zero-Suppression Rule*: we may eliminate nodes whose 1-edge leads to **0**. With these standard ZDD rules, it is not hard to see that **0** represents the empty collection of sets, while **1** represents the collection consisting of only the empty set.

ZDDs interpreted this way have a standard set of operations based on recursive definitions [10], including the union and intersection of two collections of sets, for example. These ZDD procedures form the building blocks of the Compressed-BFS algorithm, and are implemented in several publicly available BDD/ZDD libraries [16, 11]. We review informal definitions of some of the lesser-known ZDD procedures used.

- **Existential Abstraction.** Given a ZDD F, and a set of variables S, remove all occurrences of $s \in S$ from any subset in F. This can be implemented by cofactoring F with respect to $s = 1$ and $s = 0$, then forming the union of these results[11].

- **Subsumed Difference.** Given two ZDDs, F and G, form a ZDD $F \setminus_S G$ containing all subsets from F which are not subsumed by some subset in G [6].

- **PowerSet.** Given a set S, create a ZDD $\mathbf{2}^S$ which contains all subsets of S. Such a ZDD will require exactly $|S|$ nodes.

The *Subsumed Difference* [6] operation can be extended into other operations, allowing one to maintain a subsumption-free ZDD. Removal of subsumptions in some way is crucial to achieving performance within the Compressed-BFS algorithm.

2.2 Boolean Constraint Propagation

Given a valid partial truth assignment to some variables in a CNF instance, we may often easily infer additional information about any solution based on this assignment. One way of doing this is to look for *unit* clauses under this truth assignment; if a clause has all literals except one set false, then in order to satisfy this clause we must set the remaining literal true. Recursive application of this *unit clause rule* forms the basis of Boolean Constraint Propagation (BCP) within the context of DLL algorithms for satisfiability.

DLL performs a backtracking, depth-first search over the solution space of variable assignments. Thus it is natural after branching on a given variable to use BCP to force the assignment of as many variables as possible, and to immediately deduce conflicts a given assignment may create. However with regard to the Compressed-BFS algorithm, it is not as straightforward to apply this rule to prune branches of the search. We will later show how it is possible to use the unit clause rule to deduce conflicts within this framework.

3. THE COMPRESSED-BFS ALGORITHM

The use of a compressed container in algorithms for solving satisfiability has been explored in many ways before our Compressed-BFS. Combining the DP procedure with Zero-Suppressed Binary Decision Diagrams (ZDDs) was explored in the ZRes solver [7]. Using ZDDs to perform the DLL procedure was explored recently as well [3]. The idea behind the Compressed-BFS algorithm [13] was to leverage the compression power of ZDDs to mitigate the main shortcoming of Breadth-First Search: memory utilization.

Compressed-BFS proceeds analogously to a BFS. It processes variables according to a static order, and implicitly represents all promising truth assignments of a given depth d. These valid partial truth assignments are assignments to variables $x_1, x_2, \ldots, x_d$ which do not cause all literals in some clause to be assigned false. The collection of these partial truth assignments is called the front. To determine the proper state after processing variable x_{d+1}, the algorithm 'copies' the front, and modifies one copy of each assignment within this collection to reflect the additional assignment of $x_{d+1} = 1$. It modifies the other copy of the front to reflect assigning $x_{d+1} = 0$. Finally, all valid partial truth assignments arising from either of these branches might yield satisfiability, so both branches are combined into the single new front.

Rather than store explicit truth assignments, Compressed-BFS stores the subsets of open clauses corresponding to these assignments in the front. By combining this front with a new truth assignment to a single variable, the front can be advanced as described above. To update the front to reflect a truth assignment to a single variable, the effects of this truth assignment on the status of clauses must be considered. In general, an assignment to a single variable $x_i = t$ (where $t \in \{0, 1\}$) has the following effects on clauses.

- It *violates* some clauses. Let $U_{x_i,t}$ be the set of *unit* clauses for which this variable assignment causes a conflict. Then, any subset in the *front* containing some $u \in U_{x_i,t}$ must be pruned as it cannot yield satisfiability. This can be accomplished with an appropriate ZDD intersection operation.

- It *satisfies* some clauses. Let $S_{x_i,t}$ be the set of all clauses which contain a literal in $\{x_i, \overline{x_i}\}$ and $x_i = t$ makes this literal true. If these clauses were not yet satisfied, then they become satisfied by this assignment. These clauses are removed from all subsets in the *front* by ZDD *existential abstraction*.

- It *opens* some clauses. Let $A_{x_i,t}$ be the set of all clauses which were *not activated*, contain a literal in $\{x_i, \overline{x_i}\}$, and $x_i = t$ makes this literal *false*. If this literal were assigned true, the clause would not become *open* and not be needed to added to the *front*. All such

clauses $A_{x_i,t}$ are added to every subset in the *front* by the ZDD Cartesian product operation.

Determining each of these sets depends only on the particular truth assignment to $x_i = t$, and not to the internal state of the *front*. Thus, with each of these sets of clauses, an action can be taken on the entire *front*. In order to prune branches from the search containing violated clauses $U_{x_i,t}$, we build the PowerSet $\mathbf{2}^{Clauses \setminus U_{x_i,t}}$: the collection of all sets which do not contain any clauses in $U_{x_i,t}$. We then intersect this collection with the *front*, leaving only those subsets contained within $Clauses \setminus U_{x_i,t}$. Finally, we can remove subsets B which are subsumed by some other set $A \subsetneq B$ as these correspond to suboptimal partial assignments.

Initially, we have no *open* clauses, and the *front* is set to be the collection containing only the empty set, **1**. For each variable x_i, we modify one copy of the *front* as described above to reflect assigning $x_i = 1$. We modify another copy to reflect assigning $x_i = 0$. Finally, the new *front* is the union of these two, since we must consider promising branches in either case. After all variables are processed, there are two possible outcomes. If no branches lead to satisfiability, then the *front* will be empty (equal to **0**) as it contains sets of *open* clauses, each of which corresponds to a promising branch in the search. If any branches lead to satisfiability, then there will be no open clauses and the *front* will contain only the empty set (**1**). Pseudocode for the Compressed-BFS algorithm is shown in Figure 2.

ZDD algorithms depend heavily on the ordering of ZDD nodes. Because of this, our initial ordering is designed so the Cartesian Product operation takes linear time [13] by ensuring that added nodes have lower indices. Here also, Compressed-BFS's performance depends on the order in which variables are processed. Since only those clauses in the *cut* have the potential to be open clauses, we may reorder variables to reduce *cutwidth*. As a preprocessing step to Compressed-BFS, the MINCE[2] heuristic ordering is applied to attempt to reduce *cutwidth*. Reducing cutwidth is critical to Compressed-BFS's runtime: if full subsumption removal is applied, the maximum number of subsets at a given step in the front is exactly the size of the maximal anti-chain of the partially ordered set $\mathbf{2}^{Cut}$ [13].

```
Compressed-BFS(Vars, Clauses)
  Front ← 1
  for i = 1 to |Vars| do
      front' ← front
      //Modify front to reflect x_i = 1
      Form sets U_{x_i,1}, S_{x_i,1}, A_{x_i,1}
          front ← front ∩ 2^{Clauses \ U_{x_i,1}}
          front ← ∃Abstract(front, S_{x_i,1})
          front ← front ⊗ A_{x_i,1}
      //Modify front' to reflect x_i = 0
      Form sets U_{x_i,0}, S_{x_i,0}, A_{x_i,0}
          front' ← front' ∩ 2^{Clauses \ U_{x_i,0}}
          front' ← ∃Abstract(front', S_{x_i,0})
          front' ← front' ⊗ A_{x_i,0}
      //Combine the two branches via Union
      //and remove Subsumptions
      front ← front ⋃_(S) front'
  if front = 0 then
      return Unsatisfiable
  if front = 1 then
      return Satisfiable
```

Figure 2: Pseudocode for the Compressed-BFS Algorithm

3.1 Opportunistic Subsumption Removal

In Compressed-BFS, we may often reduce the overall runtime of the search procedure by investing additional runtime to eliminate subsumptions. However, the full search for subsumptions may take significant time. A simple search based on two reduction rules can be applied in linear time by a single pass over the ZDD. This opportunistic search in reality may find a significant number of subsumptions, and also preserves the utilization of autark assignments.

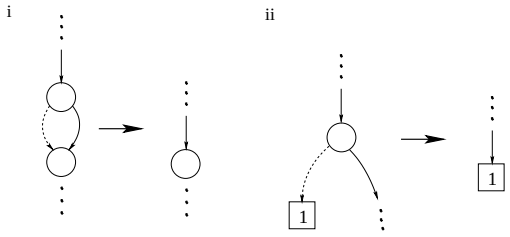

Figure 3: Two cases in which subsumptions can be easily eliminated. In (i), both children are the same. In (ii), the E-Child is 1.

The first reduction rule is based on finding subsets which differ in a single element. If there are two subsets $A \subsetneq B$ such that B has exactly one additional element, then it is not hard to see there will be some node in the ZDD which has the form shown in Figure 3:i. That is, whether the upper node is true or false will not affect the evaluation of the collection's characteristic function for sets which depend on this node. Because of the subsuming ZDD semantics this implies that we should remove all sets depending on this node, in which the node evaluates to true: this can be accomplished by setting the T-Child to **0**. However, because of the zero-suppression rule, we should simply eliminate this node as shown in Figure 3. The resulting reduction rule is the same rule used in ROBDDs: it can be said that the front in Compressed-BFS has the compression power of both ZDDs and BDDs.

The second reduction rule is based on the notion that the empty set should subsume all other sets. If the empty set is part of a collection, then it must appear as the E-Child of some node. The sub-portion of the ZDD which meets this criterion should be eliminated, as shown in Figure 3:ii. Note that this rule also encompasses more than just selection of autark assignments due to the recursively-defined semantics of ZDDs. In testing Compressed-BFS on real-world instances, memory utilization is the limiting factor rather than runtime. As a result, for this work we utilize the full search for subsumptions at each step.

4. BOOLEAN CONSTRAINT PROPAGATION

In Compressed-BFS, we store sets of open clauses in a ZDD. The main idea in augmenting this approach with Boolean Constraint Propagation (BCP) is that *conflicting* sets of clauses cannot lie within the same subset of *open* clauses in the *front*. For our purposes, a set of clauses is *conflicting* if it is possible to derive a contradiction by the unit clause rule. If all such conflicting sets of clauses are discovered, then any partial truth assignment which leaves such a set of clauses clauses open is invalid. We may prune such branches from the search by forming a ZDD containing conflicting sets of clauses. We then remove such sets from the *front* by use of the *Subsumed Difference* operator.

4.1 A Motivating Example

$$\underbrace{(\bar{a}+\bar{d})}_{1}\underbrace{(\bar{a}+\bar{c})}_{2}\underbrace{(\bar{a}+c)}_{3}\underbrace{(a+\bar{b})}_{4}\underbrace{(a+b)}_{5}$$

Here, we will consider the state of the Compressed-BFS algorithm after processing the variable a. Since the *front* consists of all sets of open

clauses, it is not hard to see that the *front* has the form shown in Figure 4. Under the assignment $a = 1$, clauses 1, 2, and 3 are open. However the assignment $a = 0$ leaves clauses 4, 5 open. Thus after variable *a* the *front* ZDD contains two subsets, $\{1,2,3\}$ and $\{4,5\}$.

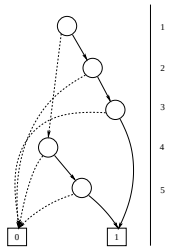

Figure 4: The *Front* after Variable *a*

After variable *a*, all clauses in this example are *unit*, having only one unassigned literal. If any of these clauses appear in some subset in the *front* then they must be *open* clauses, and imply the remaining unassigned literal. If two clauses imply literals of opposite polarity, they cannot appear in the same subset. This basic rule forms the basis of a "depth 1" BCP procedure: we can form the *Cartesian Product* of the set of clauses implying some literal l and the set of clauses implying $\bar{l}$. Each pair in this product will contain a clause implying l and one implying $\bar{l}$, and thus the *Cartesian Product* only contains conflicting sets of clauses. Any subset in the *front* which contains such a pair of clauses can be pruned. Notice that sets of conflicting clauses at a given step in the algorithm are independent of the front.

For the given example, clause 2 implies the literal $\bar{c}$, while clause 3 implies the literal *c*. Thus, any subset in the *front* containing $\{2,3\}$ can be pruned. Similarly because of the literal *b*, any subset containing $\{4,5\}$ can be pruned. The ZDD containing these subsets is shown in figure 5. These implied conflicts would be removed via the *Subsumed Difference* operator. In this case the resulting *front* will be empty, since each subset of the front is subsumed by some subset in the Conflict ZDD, and we can conclude the formula is unsatisfiable. To contrast this, the original Compressed-BFS algorithm would be forced to process variables *b* and *c* before determining that the formula was unsatisfiable.

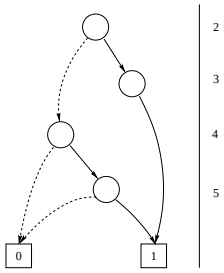

Figure 5: Conflicting Sets of Clauses after Variable *a*

4.2 The BCP Procedure

When finding implications at a given stage, Compressed-BFS is not limited to finding the "depth 1" implications as shown above. Rather, such implications can be propagated, possibly deriving additional sets of violated clauses. The premise of a general procedure to find all conflicting sets of clauses is to consider the effects of assigning some variable *v* either *true* or *false* and keeping track of which clauses are violated by doing this. By choosing *v* only out of literals which are implied by some *unit* clause, we effectively capture only the effect of implications on our formula. In the "depth 1" BCP mentioned above, these violated clauses were stored in sets. The Cartesian product of such sets gives conflicting pairs of clauses. Rather than simply find a set of clauses which is violated by this single assignment, we can recursively build a ZDD taking into account multiple assignments. Simple pseudocode for such a recursive search is shown in Figure 6. The expressiveness of the ZDD data structure helps make such a recursion possible.

```
GetConflictZDD(Formula F', Integer Var)
  foreach clause C ∈ F'
      if C has no literals (after the cut)
          //Then C is a violated clause
          ViolCls ← ViolCls ⋃ C.

  //Find the set of variables implied by some unit clause
  IVars ← ImpliedVars( Units(F') )

  //Find the lowest index variable v implied by some
  //unit clause, such that v > Var
  v_low ← UpperBound(IVars, Var)
  if no such v_low exists
      return ViolCls
  ConflZdd ← ViolCls
  //Iterate over all implied variables ≥ v
  forall v ∈IVars such that v ≥ v_low
      //Modify F' to reflect assignment, and recurse
      Z1 ← GetConflictZDD(Assign(F', v = 1), v)
      Z0 ← GetConflictZDD(Assign(F', v = 0), v)
      Z ← Z0 ⊗ Z1
      ConflZDD ← ConflZDD ⋃ Z
  return ConflZDD
```

Figure 6: Constraint Propagation Pseudocode

At any stage in the recursion, the procedure *GetConflictZDD* only returns violated clauses or conflicting sets of clauses. This procedure for finding all conflicts after a given step in the search is based on conditioning on a given variable. Since the variable must be either true or false, we must encounter conflicts found in one of these two branches. Initially, one would invoke *GetConflictZDD* by passing it a simplified formula F' from the original CNF instance. Here we may remove all clauses before the *cut* as well as all literals before the *cut* from remaining clauses. The procedure *Assign* works by essentially removing all clauses in F' which are satisfied by the assignment to some variable *v*, and removing all literals of *v* from clauses which are not satisfied.

The procedure *GetConflictZDD* works as follows. It first finds all violated clauses. It then branches over the implications of all *unit* clauses. For each such variable *v* implied by some unit clause, it forms the ZDD $Z1$ of all conflicts when $v = 1$. It similarly forms the ZDD $Z0$ of all conflicts when $v = 0$. Note that since this variable was implied by some unit clause *c*, then *c* necessarily appears as a single element in either $Z1$ or $Z0$. Since *v* must be either true or false, we must encounter conflicts in either $Z1$ or $Z0$. Then, any subset in the *front* which is subsumed by some subset in the *Cartesian Product* $Z0 \otimes Z1$ cannot yield satisfiability as it has elements from both $Z1$ and $Z0$.

Finally, *GetConflictZDD* takes the union of all such branches. The subsuming semantics of ZDDs used in Compressed-BFS also apply here: we may remove subsumed sets from the conflict ZDD. The only remaining point of the procedure is the use of the Integer *Var* in the search. This is used to break the symmetry which arises during the search by imposing an ordering on branches. Thus we will never attempt to assign, e.g.

$c = 0$ then $d = 1$ as well as recursing on $d = 1$ then $c = 0$. Since the combined effect of these assignments is the same, we can impose an ordering on variables being assigned and eliminate such redundancy. It should be noted that the ordering used on the set of implied variables in general *cannot* simply by the variable ordering used to process variables within Compressed-BFS. This is because after branching on variable *v*, unit clauses may in fact imply some variable which occurs before *v* with this ordering. Instead, we must order the search based on the order in which clauses in F' become unit, and new variables are implied. These new implications should be inserted later in this ordering, to ensure that we maintain completeness.

4.3 Extending BCP

Although the procedure presented above will find all possible conflicting sets of clauses at a given stage in the algorithm, it may take significant time to perform such an exhaustive search if there are many implications. Another difficulty is that deeper searches can only find larger sets of conflicting clauses. Larger sets are less likely to subsume subsets in the *front*, and are thus of limited benefit.

Both problems are addressed by bounding the depth used in the recursion to a given level. This places a bound on the number of clauses within any set of conflicting clauses the procedure finds, increasing the chance that these sets subsume some set in the *front*. Such a restriction will also help limit the runtime of such a search.

In addition, the BCP-based search for conflicts at variable *k* will share much similarity with the search at variable $k+1$, limiting the effectiveness of such a search. Here we apply BCP only every $2d$ steps, where *d* is the depth to which we perform the search. In addition we limit propagation of assignments to those 'near the cut' by a factor of $3d$. Finally, during the first and last few steps of Cassatt's search, the front tends to already be quite small, and additional search will be superfluous. To help compensate for this, here we do not apply BCP during the first and last 10% of variables. However when BCP should be applied is considered a tunable parameter of our search.

5. EMPIRICAL RESULTS

Cassatt, our implementation of the Compressed-BFS algorithm, is written entirely in C++ using the CUDD package [16]. In addition, we use the existential abstraction procedure from the *Extra* library [11]. To obtain these results, we disabled reordering in CUDD. Cassatt runtimes do not include time required to obtain the MINCE variable ordering or I/O time. Other solvers were run with default configurations.

Our results fall into two categories. First, on instances which lack a great deal of structure such as randomized instances and many real-world instances the original Cassatt algorithm performs relatively poorly. It is on these instances which the addition of BCP gives an increase in performance. However this increase is modest, and on these instances Cassatt is not competitive with DLL based solvers. We compare Cassatt without BCP to Cassatt with bounded depth BCP. In Figure 7 results are shown on instances from the DIMACS benchmark suite. The results in Figure 7 were obtained on an AMD Athlon 1.2GHz machine with 1GB DDR RAM running Linux. Runtimes of DLL-solvers are not included as they solve all instances quickly.

On `aim` instances, the addition of BCP seems to help, especially as problem size increases. For `aim-200` and `aim-100` instances, depth 3 BCP appears to give the best performance. Instances which were not solved within a 250-second timeout limit were treated as if they took 250 seconds; this slightly skews the shown results in favor of Cassatt without BCP as the addition of BCP allows Cassatt to solve more instances.

Our second class of results is over instances with large amounts of structure such as pigeonhole instances. These instances are often designed to defeat DLL-based solvers or resolution in general. Here Cassatt performs extremely well and the addition of BCP only hinders the algorithm. However these instances are solved so quickly that any addition to the algorithm will likely have this result.

The XOR-Chain family of benchmarks are known to be difficult for solvers based on DLL algorithms while easy for some other methods of solving SAT [1]. Here we show these are easy for Cassatt as well. We report results of zChaff Z2001.2.17 to show typical performance of tuned DLL-based solvers. [1] In Figure 8 runtimes for Cassatt and zChaff

XOR-C	S/U	Cassatt	BCP 2	BCP 3	BCP 4	zChaff
1_16	UNS	0.01	**0**	0.01	0.01	0.99
1_24	UNS	**0**	0.01	0.02	0.03	8.35
1_32	UNS	**0.01**	0.02	0.03	0.07	59.72
1_36	UNS	**0.02**	0.03	0.03	0.06	1088.95
1_40	UNS	**0.01**	0.03	0.06	0.18	MEM-OUT
1_64	UNS	**0.02**	0.06	0.09	0.15	MEM-OUT
1_128	UNS	**0.12**	0.2	0.35	0.59	MEM-OUT
1.1_16	UNS	**0.01**	**0.01**	0.02	0.02	0.99
1.1_24	UNS	**0**	0.02	0.03	0.02	8.33
1.1_32	UNS	**0.01**	0.03	0.03	0.06	92.54
1.1_36	UNS	**0.02**	**0.02**	0.03	**0.02**	MEM-OUT
1.1_40	UNS	**0.01**	0.03	0.03	0.05	MEM-OUT
1.1_64	UNS	**0.03**	0.06	0.1	0.15	MEM-OUT
1.1_128	UNS	**0.08**	0.22	0.42	0.47	MEM-OUT
2_16	UNS	**0.01**	0.02	**0.01**	0.03	0.11
2_24	UNS	**0**	0.01	0.02	0.04	11.89
2_32	UNS	**0.01**	0.02	0.04	0.04	113.94
2_36	UNS	**0.02**	0.04	0.04	0.07	305.16
2_40	UNS	**0.01**	0.02	0.04	0.06	1540.36
2_64	UNS	**0.03**	0.08	0.13	0.21	MEM-OUT
2_128	UNS	**0.1**	0.22	0.42	0.64	MEM-OUT

Figure 8: Runtimes for the XOR-Chain family

were obtained on a 2.0GHz Pentium 4 Xeon with 1.0GB RAM. As mentioned, on these structured instances BCP does not provide an improvement, but the original runtimes are so low that should any improvement exist, it would be difficult to detect empirically.

A similar situation occurs for instances based on FPGA switchbox routing [4]. Figure 9 shows results for Cassatt on instances based on

FPGA	S/U	Cassatt	BCP 2	BCP 3	BCP 4	zChaff
10_11	UNS	**0.04**	0.12	0.45	1.18	>250
10_12	UNS	**0.05**	0.14	0.35	0.96	>250
10_13	UNS	**0.03**	0.15	0.59	2.01	>250
10_15	UNS	**0.09**	0.34	1.31	6.39	>250
10_20	UNS	**0.24**	0.7	2.82	15.1	>250
11_12	UNS	**0.06**	0.16	0.59	1.1	>250
11_13	UNS	**0.04**	0.15	0.74	2.97	>250
11_14	UNS	**0.04**	0.21	0.98	4.09	>250
11_15	UNS	**0.06**	0.24	1.06	5.43	>250
11_20	UNS	**0.1**	0.51	3.3	20.68	>250
10_8	SAT	**0.03**	0.07	0.28	2.6	2.13
10_9	SAT	**0.06**	0.13	0.36	1.24	2.01
12_8	SAT	**0.06**	0.12	0.37	2.03	>250
12_9	SAT	**0.12**	0.19	0.53	2.36	104.7
12_10	SAT	**0.15**	0.26	0.87	3.97	>250
12_11	SAT	**0.07**	0.2	0.83	4.97	>250
12_12	SAT	**0.52**	0.67	1.55	5.68	132.91
13_9	SAT	**0.35**	0.44	0.8	2.79	191.63
13_10	SAT	**0.71**	0.84	1.43	5.47	66.3
13_11	SAT	**1.61**	1.8	2.4	4.47	>250
13_12	SAT	**2.66**	2.88	3.62	7.99	>250

Figure 9: Runtimes for instances from FPGA Switchbox Routing

FPGA switchbox routing [4]. Again, runtimes for Cassatt on these in-

[1] At the SAT2002 solver competition, zChaff outperformed all other solvers on the **x2*** family of instances and performed competitively on **x1*** and **x1.1***.

Benchmark Family	#	S/U	Cassatt		+ BCP Depth 2		+ BCP Depth 3		+ BCP Depth 4	
			%Sol	Avg	%Sol	Avg	%Sol	Avg	%Sol	Avg
aim-200-1_6-yes1	4	SAT	75	122.95	75	117.7	75	**113.95**	75	119.94
aim-100-1_6-no	4	UNS	100	.56	100	1.165	100	.6525	100	.885
aim-100-1_6-yes1	4	SAT	100	.06	100	.09	100	.13	100	1.08
aim-100-2_0-no	4	UNS	25	187.51	25	187.52	25	187.53	25	187.52
aim-100-2_0-yes1	4	SAT	100	33.6	100	31.8	100	29.1	100	31.5
aim-100-3_4-yes1	4	SAT	0	>250	25	237.4	25	231.25	25	218.16
aim-100-6_0-yes1	4	SAT	100	32.5	100	19.0	100	17.8	100	37.2
aim-100*	24	-	70.83	84.04	75	79.5	75	**77.74**	75	79.39
aim-50-1_6-no	4	UNS	100	0.02	100	0.02	100	0.04	100	0.07
aim-50-1_6-yes1	4	SAT	100	0.02	100	0.025	100	0.03	100	0.06
aim-50-2_0-no	4	UNS	100	0.14	100	0.14	100	0.18	100	0.33
aim-50-2_0-yes1	4	SAT	100	0.04	100	0.05	100	0.08	100	0.16
aim-50-3_4-yes1	4	SAT	100	0.54	100	0.45	100	0.58	100	1.3
aim-50-6_0-yes1	4	SAT	100	0.29	100	0.30	100	1.23	100	8.2
aim-50*	24	-	100	0.18	100	**0.17**	100	0.355	100	1.69
dubois*	13	UNS	100	**0.01**	100	0.02	100	0.02	100	**0.01**
pret*	8	UNS	100	**0.016**	100	0.018	100	0.02	100	0.02
par16*	5	SAT	80	**85.52**	60	129.19	60	131.338	60	136.75
par16-c*	5	SAT	60	**152.42**	60	154.67	60	155.26	60	159.00
par8*	5	SAT	100	0.71	100	**0.488**	100	0.89	100	2.04
par8-c*	5	SAT	100	**0.026**	100	0.058	100	0.128	100	0.45

Figure 7: Instances from the DIMACS benchmark suite

stances are low, and BCP does not further improve them. Results in Figure 9 were obtained on an AMD Athlon 1.2GHz machine with 1GB RAM.

6. CONCLUSIONS AND ONGOING WORK

Our work extends the Compressed-BFS algorithm by recursive application of the *unit clause rule*. The original Compressed-BFS's runtime (with opportunistic subsumption removal) is polynomially dependent on the size of the compressed representation it uses. However, (unless $P = NP$) this representation will have worst-case superpolynomial size. By the addition of BCP to this algorithm, we attempt to reduce the size of the representation at each step by investing additional runtime to derive conflicts in the search.

The compressed-BFS algorithm was developed to counter the shortcomings of a traditional BFS. By using an implicit representation, then in certain cases Compressed-BFS avoids such an explosion in memory utilization. However, the main bottleneck for this algorithm on real-world instances is still memory utilization. Thus the addition of BCP can be viewed as a measure to counter the shortcomings of the original Compressed-BFS algorithm. Since the runtime of Compressed-BFS depends on the representation size, an investment of run-time in reducing the ZDD size is likely to pay off by causing later reductions.

The search for conflicts by use of the *unit clause rule* need not be complete. As a result, there is room for improvement over the straightforward complete search. Heuristic and randomized approaches can be applied, with the aim of quickly covering a significant fraction of the search for conflicting sets of clauses. In addition, making optimal use of BCP within Compressed-BFS is still an area we consider. Empirically, DLL-based solvers have difficulty on some families which Cassatt can quickly solve. However the converse is also true, as several instances from the DIMACS suite are unsolvable with Cassatt. On several instances, use of BCP does reduce the runtime of Cassatt. On highly structured instances the compression power of the ZDD allows Cassatt to efficiently determine satisfiability. However on these instances the addition of BCP does not help the algorithm.

Our ongoing work proceeds in several directions. We hope to improve performance of our Compressed-BFS+BCP implementation further via tuning. We are also studying modifications of well-known SAT solvers that are required to produce resolution proofs of unsatisfiability rather than just a negative answer. Finally, we hope to modify traces saved by Compressed-BFS so that they form the basis of verifiable proofs for unsatisfiable instances.

7. REFERENCES

[1] "SAT2002 Solver and Benchmark Competition". http://www.satlive.org/SATCompetition/index.jsp.

[2] F. Aloul, I. Markov, and K. Sakallah. "Faster SAT and Smaller BDDs via Common Function Structure". *Proc. ICCAD*, 2001.

[3] F. Aloul, M. Mneimneh, and K. Sakallah. "Backtrack Search Using ZBDDs". *IWLS*, 2001.

[4] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. 'Solving Difficult SAT Instances in the Presence of Symmetry". *39th ACM/IEEE DAC*, 2002. http://www.eecs.umich.edu/~faloul/benchmarks.html.

[5] P. Beame and R. Karp. "The Efficiency of Resolution and Davis-Putnam Procedures". to appear SIAM Journal on Comp.

[6] P. Chatalic and L. Simon. "Multi-Resolution on Compressed Sets of Clauses". *Proc. of 12th International Conference on Tools with Artificial Intelligence (ICTAI-2000)*, November 2000.

[7] P. Chatalic and L. Simon. "ZRes: the old DP meets ZBDDs". *Proc. of the 17th Conf. of Autom. Deduction (CADE)*, 2000.

[8] M. Davis, G. Logemann, and D. Loveland. "A Machine Program for Theorem Proving". *Comm. ACM*, 5:394–397, 1962.

[9] A. Haken. "The Intractability of Resolution". *Theoretical Computer Science*, 39:297–308, 1985.

[10] S. Minato. "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems". *30th ACM/IEEE DAC*, 1993.

[11] A. Mishchenko. "EXTRA v. 1.3: Software Library Extending CUDD Package: Release 2.3.x". http://www.ee.pdx.edu/~alanmi/research/extra.htm.

[12] M. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". *Proc. of IEEE/ACM DAC*, pages 530–535, 2001.

[13] D. Motter and I. L. Markov. "A Compressed Breadth-First Search for Satisfiability". *Proc. 4th Workshop on Algorithm Engineering and Experiments*, 2002.

[14] D. Motter and I. L. Markov. "On Proof Systems Behind Efficient SAT Solvers". *Fifth International Symposium on Theory and Applications of Satisfiability Testing*, 2002.

[15] J. Marques Silva and K. Sakallah. "GRASP: A New Search Algorithm for Satisfiability". *ICCAD* , 1996.

[16] F. Somenzi. "CUDD: CU Decision Diagram Package Release 2.3.1". http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.

[17] A. Urquhart. "Hard Examples for Resolution". *Journal of the ACM*, 34(1):209–219, 1987.